NORTH-HOLLAND

A CONFLUENT SEMANTIC BASIS FOR THE ANALYSIS OF CONCURRENT CONSTRAINT LOGIC PROGRAMS

MICHAEL CODISH, MORENO FALASCHI,
KIM MARRIOTT, AND WILLIAM WINSBOROUGH

▷ The standard operational semantics of concurrent constraint logic languages is not confluent in the sense that different schedulings of processes may result in different program behaviors. While implementations are free to choose specific scheduling policies, analyses should be correct for all implementations. Moreover, in the presence of parallelism, it is usually not possible to determine how processes will actually be scheduled. Efficient program analysis is therefore difficult as all process schedulings must be considered. To overcome this problem, we introduce a confluent semantics which closely approximates the standard (nonconfluent) semantics. This semantics provides a basis for efficient and accurate program analysis for these languages. To illustrate the usefulness of this approach, we sketch analyses based on abstract interpretations of the confluent semantics which determine if a program is suspension- and local suspension-free. © *Elsevier Science Inc., 1997* ◁

1. INTRODUCTION

Concurrent constraint logic programming [24, 26] is a programming paradigm based on logic programming with mechanisms for concurrency. In recent years, there has

Address correspondence to Michael Codish, Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, PoB 653 Beer-Sheba, Israel. E-mail: codish@cs.bgu.ac.il; Moreno Falaschi, Dipartimento di Matematica e Informatica, Via delle Scienze, 206, Udine, Italy. E-mail: falaschi@dimi.uniud.it; Kim Marriott, Department of Computer Science, Monash University, Clayton 3168, Vict., Australia. E-mail: marriott@cs.monash.edu.au; William Winsborough, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, U.S.A. E-mail: winsboro@cse.psu.edu.

Received March 1995; accepted December 1995.

655 Avenue of the Americas, New York, NY 10010

0743-1066/97/$17.00
PII S0743-1066(96)00013-1

been a growing interest in both theoretical as well as practical applications of these languages. However, before their full potential can be realized, there is a need for powerful tools to provide efficient and accurate program analyses which can be used by compilers to produce more efficient target code and by programmers to detect (synchronization) errors in code. The main contribution of the present paper is a semantic basis for the development of such analyses.

The computational model of concurrent constraint logic programming is based on *constraints* and an entailment or implication relation between these. Processes interact through a common store of constraints. Communication is achieved by *telling*, that is, adding, a given constraint to the store (asynchronous message send), and by *asking* whether the store entails a given constraint (asynchronous message receive). Nondeterminacy arises in two ways: because there is a choice of which clause to reduce a process with, and because of different process schedulings.

Confluence, that is, independence of scheduling of reductions, is an important and desirable semantic property of declarative languages. In particular, it allows a program to be understood using any convenient scheduling as other schedulings lead to "isomorphic" results. For example, confluence holds in the lambda calculus because of the Church–Rosser property, and it holds in logic programming because of the Switching Lemma [15]. In the context of concurrency, confluence is an even more desirable property [18] as concurrent programs are notoriously difficult to reason about and to analyze and, without confluence, all possible scheduling rules, and hence interleavings, must be considered. In concurrent languages, we are also interested in nonterminating computations. In this case, we will understand confluence as independence of scheduling for all "fair" reduction sequences in the sense that the possible outcomes of the computations are the same. However, because of the interaction between nondeterminism and synchronization, confluence does not hold for many concurrent languages. In particular, it does not hold for concurrent constraint logic languages, the class of languages we are interested in analyzing. In these languages, implementations are free to choose a particular process scheduling policy. However, as shown in Example 2.1 (Section 2), the standard operational semantics is not confluent with respect to different schedulings. Efficiency is therefore problematic in analyses which are directly based on the standard semantics as they must consider all possible process schedulings so as to ensure correctness for any implementation.

For this reason, we introduce a confluent semantics which approximates the standard (nonconfluent) semantics of concurrent constraint logic languages. We propose this semantics as a basis for accurate and efficient program analysis. Correctness of such analyses holds because the confluent semantics approximates the standard semantics in the sense that suspension in the usual semantics implies suspension in the confluent semantics. Accuracy holds because the standard semantics is "nearly" confluent—in fact, for deterministic programs and programs without synchronization, the two semantics coincide—and so the approximation is very close. Finally, because of confluence, an analysis based on this semantics need only be proven correct for a single scheduling rule. This provides for accuracy as the analysis can choose a scheduling which gives the most precise answer, and also provides for efficiency as there is no need to examine the potentially exponential or even infinite number of different but "isomorphic" reduction sequences corresponding to other schedulings.

To illustrate the usefulness of the confluent semantics as a basis for abstract interpretation, we sketch analyses for detecting "suspension" and "local suspension." Local suspension occurs when a process in a system can never be reduced as it requires input from other processes in its environment to continue. Suspension is an acute form of local suspension in which the computation halts as no process in the system can be scheduled.

Most of the paper discusses a language with atomic publication [23, 26]. Atomic publication requires that the *tell* constraints of a clause be consistent with the current constraint store for the clause to be used in process reduction. By contrast, eventual publication does not consider the *tell* component when selecting a viable clause, but requires only that the clause's *ask* constraint be entailed by the constraint store. We formalize eventual publication and summarize corresponding results. The properties of the eventual publication language's confluent semantics are stronger: assuming a fair computation rule, each nonfailed derivation in the usual semantics has a corresponding isomorphic derivation in the confluent semantics. Since the results for the atomic case are more difficult, we emphasize them in this paper.

The rest of this paper is organized as follows. In the next section, we give several motivating examples which illustrate the main ideas. Section 3 makes precise the usual operational semantics of concurrent constraint logic programs. In Section 4, we discuss several different types of suspension which are of interest in the context of concurrent constraint logic languages. Section 5 introduces a confluent semantic basis for the analyses which are described in Section 6. Section 7 gives related results for languages with eventual publication. Section 8 discusses the role of types for suspension analyses. Section 9 reviews related work, and Section 10 concludes. The Appendix contains proofs of the main theorems. A preliminary version of this paper appeared in [5].

2. MOTIVATING EXAMPLES

The operational semantics of concurrent constraint logic programs is formalized in the next section. The intuitive idea is that processes are identified with atoms which communicate and synchronize through a common store of constraints. Computation starts with an initial environment or "state" containing a set of processes and the current constraint store. Computation proceeds by repeatedly using clauses in the program to *reduce* processes in the state. Reduction using the (renamed) clause $C = H : -Ask : Tell \mid B$ can occur if H matches the process, and the current constraint implies Ask and is consistent with $Tell$. Reduction occurs by replacing the process by the body of the clause, B, and adding $Tell$ to the current store. Thus, processes communicate by "telling" a constraint to the store, and synchronize by "asking" the store if a particular constraint is implied by it. Reduction continues until there are no processes left, in which case the current constraint is an *answer* of the original state, or until no process can be reduced. A process in a state is *stuck* if: (1) it cannot be reduced by any clause, and (2) at least one of the clauses defining the process has constraints which are consistent with the current store. The initial state *suspends*, or *leads to suspension* if some sequence of reductions leads to a state in which all processes are stuck.

The operational semantics for a program and initial state is given as a *transition system* which is a graph with nodes labeled by states and arcs indicating reductions,

where the initial state is the "source." We adopt the convention of underlining the scheduled atom in a state and labeling the arc with the clause with which it is reduced. Different transition systems result from different process schedulings. In the following examples, we take (conjunctions of) syntactic equations over the Herbrand universe as constraints. We follow the convention that $u, v, w, x, y, z, \ldots$ are variables, a, b, and c are constants, and [ ] is the list constructor.

Example 2.1. Consider the following program P and state $s = \langle p(x), q(y); x = y \rangle$.

```
(1) p(x) :- x=a : true | s(z).     (4) q(y) :- true : y=a | true.
(2) p(x) :- x=a : true | true.     (5) q(y) :- true : y=b | true.
(3) p(x) :- true : x=b | true.     (6) s(z) :- z=a : true | true.
```

Using a left-to-right scheduling rule, the program behaves deterministically as $p(x)$ can reduce only with clause (3). Thus, under this scheduling rule, the state s does not suspend and computation terminates with answer $x = b \wedge y = b$. The behavior, however, is radically different if a right-to-left scheduling rule is applied. In this case, the program is no longer deterministic. It has three possible reduction sequences, one which leads to suspension (as the atom $s(z)$ is stuck), and others which give answers $x = a \wedge y = a$ and $x = b \wedge y = b$, respectively. Figures 1 and 2 illustrate transition systems for P and s with left-to-right and right-to-left scheduling rules, respectively.

This example demonstrates that analyses based directly on a standard operational semantics must consider all possible schedulings. In order to achieve independence of scheduling, we must guarantee that whenever a process is enabled, it can make the same choices regardless of when it is scheduled. There are two issues to consider: (1) an enabled process which is scheduled later in the standard semantics may reduce with more clauses, and (2) an enabled process in the standard semantics may later become disenabled (and not fail) (see Example 2.2). The basic idea of the confluent semantics is to separate synchronization from nondeterminism by interpreting synchronization at the procedure level instead of at the clause level. Namely, if every instance of an atom is either: (a) inconsistent with all clauses (i.e., its reduction would fail), or (b) can reduce with some clause in the standard semantics, then it can reduce with all consistent clauses in the confluent semantics. By considering all instances, we are sure that enabled processes do not suspend later (and hence can be scheduled); by reducing all consistent clauses, we are sure that we consider all potential choices at the time of scheduling.

Figure 3 illustrates the intuition behind basing analyses on a confluent semantics. The figure shows the transition system resulting from using a left-to-right scheduling with the confluent semantics for the program from Example 2.1. Solid arcs indicate

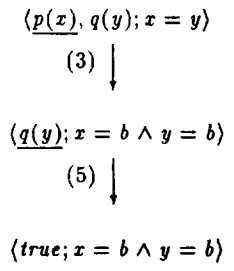

FIGURE 1. Left-to-right scheduling—standard semantics—Example 2.1.

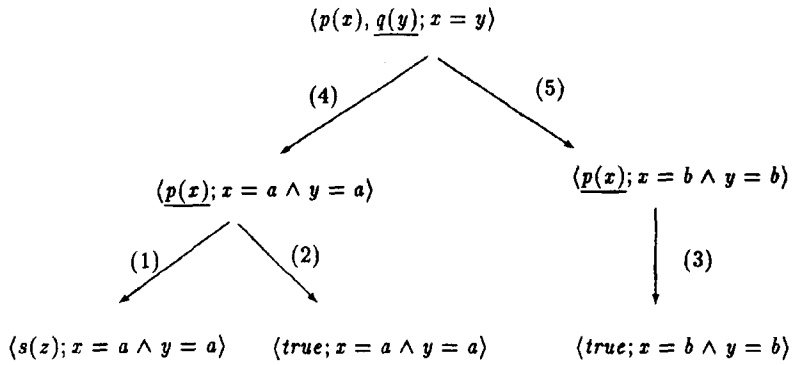

FIGURE 2. Right-to-left scheduling—standard semantics—Example 2.1.

the reductions of the standard semantics, while dotted arcs indicate additional reductions introduced by the confluent semantics.

The intuition behind basing analyses on a confluent semantics is that any reduction sequence in the standard semantics has an "isomorphic" reduction in the confluent semantics in which the reduction order is possibly changed and suspensions occur possibly sooner. Thus, an analysis based on the confluent semantics inherits the ability to detect suspension by considering only the transition system for a single scheduling rule as a program is suspension-free for all schedulings in the standard semantics if it is suspension-free for any one scheduling rule in the confluent semantics.

Example 2.2. As another example, consider the following program P and state $s = \langle p(x_1, y_1), q(x_2, y_2); x_1 = x_2 \wedge y_1 = y_2 \rangle$.

```
(1) p(x,y) :- true : x=a | true.  (3) q(x,y) :- true : x=b | true.
(2) p(x,y) :- y=b : true | true.  (4) q(x,y) :- x=a : true | true.
```

With a left-to-right scheduling rule, the standard computation is successful and gives $x_1 = a$. With a right-to-left scheduling, the (standard) computation suspends after reducing $q(x_2, y_2)$ with the third clause. This illustrates that independence of scheduling for $p(x_1, y_1)$ does not hold in the standard semantics. In the confluent semantics, both left-to-right and right-to-left schedulings will reduce $q(x_2, y_2)$ because it is not the case that all instances of $p(x_1, y_1)$ which do not fail can reduce with

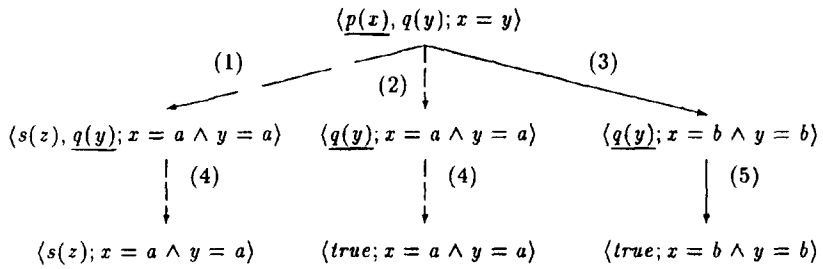

FIGURE 3. Left-to-right scheduling—confluent semantics—Example 2.1.

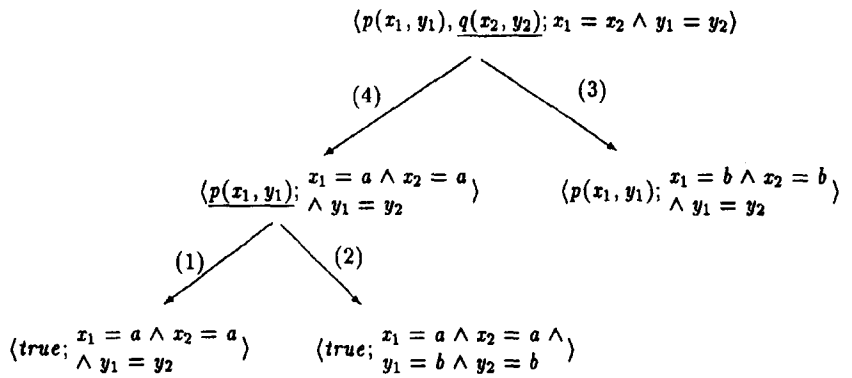

FIGURE 4. Confluent transition system—Example 2.2.

some of the clauses. Figure 4 illustrates a confluent transition system for P and s. As this transition system contains a suspended process, namely $p(x_1, y_1)$, we know that some computation rule in the standard semantics may lead to suspension.

The previous examples suggest that the confluent semantics is a good basis for suspension analyses. In practice, however, we would like a more refined analysis which also detects "local suspension," that is, when some process can never be reduced.

Example 2.3. The initial state $\langle clock(x), q(y); x = tick\rangle$ with the program

```
(1) clock(x) :- x=tick : y=tock | clock(y).
(2) clock(x) :- x=tock : y=tick | clock(y).
(3) q(x) :- x = a : true | true.
```

does not suspend as *clock* can always be scheduled, but does have local suspension because the process $q(y)$ can never be reduced.

Local suspension analyses can also be based on the confluent semantics. The following example illustrates the intuition behind basing such analyses on the confluent semantics. The program exemplifies a common technique called *incomplete messages* for specifying two-way communication between a pair of processes. Incomplete messages are messages that contain variables to be instantiated by the receiver and then read by the sender.

Example 2.4. Consider the following program P with state $s = \langle p(x_1), c(x_2); x_1 = x_2\rangle$.

```
(1) p(x) :- true : x = [msg(y)| x1] | read(y), p(x1).
(2) p(x) :- true : x = [ ] | true.
(3) c(x) :- x  = [msg(y)| x1] : true | write(y), c(x1).
(4) c(x) :- x  = [ ] : true | true.
(5) read(y) :- y = a : true | true.
(6) write(y) :- true : y = a | true.
```

For any reasonable scheduling, no process will be stuck forever. However, an "unfair" scheduling, which only schedules $p(x_1)$, will produce *read* processes that will remain forever stuck because they require $c(x_2)$ to be scheduled. For this reason,

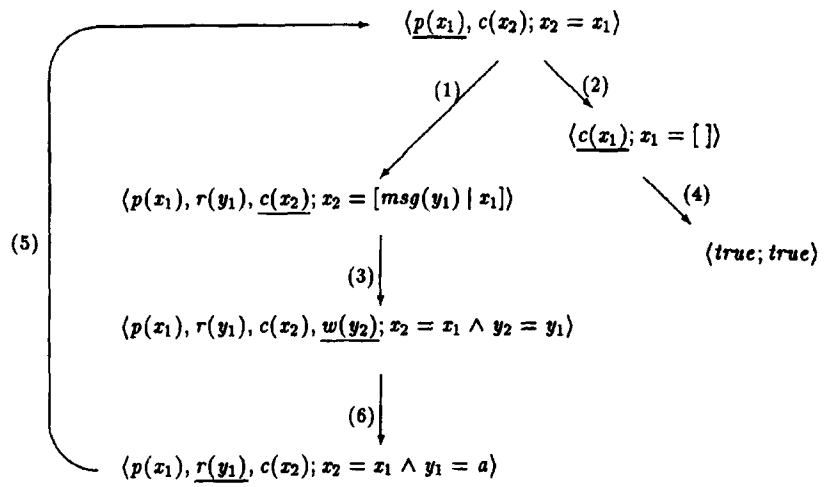

FIGURE 5. Fair confluent transition system for Example 4.

when considering local suspension, we restrict our attention to fair schedulings in which any process which is not stuck is eventually reduced. Now, independence of process scheduling holds for all fair infinite reduction sequences in the confluent semantics. Thus, to show that a program and state are free of local suspension, we need only construct a transition system based on the confluent semantics, which is local suspension-free and which has fair scheduling. Such a transition system for P is given in Figure 5, implying that P and s do not locally suspend for any implementation of the standard semantics. In this case, a suitable (and finite) transition system is obtained by consecutively scheduling the processes `p`, `c`, `read`, and `write`.

Example 2.4 also demonstrates the efficiency obtained by considering only one scheduling policy, as there are an infinite number of different process schedulings.

The actual analyses for suspension and local suspension which we develop are simple abstractions from the confluent semantics in which the constraints are replaced by constraint descriptions and similar states are merged into the same state. The analyses work by constructing a confluent transition system over these "abstract states" with the desired property. We formalize the ideas behind these examples in the remainder of the paper.

3. THE STANDARD OPERATIONAL SEMANTICS

This section presents an operational semantics for concurrent constraint logic programs which formalizes the one described in the previous section. The definitions are parametric with respect to the underlying constraint system. Moreover, almost the same definitions apply when defining the confluent semantics and the suspension analyses.

We let Con_C be a fixed set of (concrete) constraints that is closed under conjunction and existential quantification. Elements of Con_C are regarded modulo logical equivalence. Typical examples of Con_C are constructed from syntactic equations over the Herbrand universe, or linear arithmetic equalities and inequalities. We let *fail* denote the unsatisfiable constraint and *true* the always satisfiable constraint. We

write $\theta \leq \theta'$ if θ logically implies θ'. Thus, Con_C is a lattice ordered by $\leq$ with bottom element *fail* and top element *true*. For a finite set V of variables and $\theta \in Con_C$, we use $\exists_V \theta$ as shorthand for the constraint $\exists v_1 \exists v_2 \cdots \exists v_n \theta$ where $\{v_1, \ldots, v_n\} = V$. We use $\bar{\exists}_s \theta$ as shorthand for $\exists_{vars(\theta) \setminus vars(s)} \theta$ where $vars(s)$ denote the set of variables occurring in syntactic object s. Intuitively, $\bar{\exists}_s \theta$ restricts the constraint θ to the variables in syntactic object s.

We adopt a slightly nonstandard notion of goals. The standard notion of a "goal" or "environment" is captured by a state that consists of a *goal* and the current constraint store. Here, a *goal* is a (possibly empty) set of atoms that do not share variables. The empty goal is denoted by *true*, and the set of goals is denoted by *Goal*. Consequently, interprocess communications are always specified in a constraint, and as a result, it is sufficient to base analyses on descriptions of constraints. Moreover, as atoms in a goal are renamed apart, if we assume that every atom contains at least one variable, goals may be viewed as sets instead of the more cumbersome multiset notation.

Let *Con* be a lattice. The *set of states* constructed from *Con* is defined by $State = Goal \times Con$. Associated with *State* are the projections $goal : State \rightarrow Goal$ and $con : State \rightarrow Con$ defined by $goal\langle g, \theta \rangle = g$ and $con\langle g, \theta \rangle = \theta$. The *concrete states* constructed from Con_C are denoted $State_C$. We will also be interested in states constructed from abstract constraints which are descriptions of concrete constraints.

A *program* is a finite set of guarded clauses. A *guarded clause* (or *clause*) is a formula of the form $A : -Ask : Tell \mid B$ where A is an atom, called the *head*, *Ask* and *Tell* are unquantified concrete constraints, and B, called the *body*, is a goal. We let *Atom* denote the set of atoms, and *Clause* the set of clauses. The set of programs is denoted *Prog*.

A *(variable) renaming* is a bijection on the set of variables. The set of renamings is denoted by *Ren*. Renamings are naturally extended to mappings from terms to terms, etc. Often, we will be interested in terms, atoms, clauses, constraints, or states modulo variable renaming. We write $p \sim q$, and say that p is a *renaming* of q if there is a renaming ρ such that $\rho(p) = q$. For program P and a syntactic object s, $C \ll_s P$ denotes that C is a renaming of a clause in P such that $vars(C) \cap vars(s) = \emptyset$.

The operational semantics is given in terms of reductions between states. This is modeled using a *try* function and a *resolve* function. The *try* function indicates whether an atom (in a state) can be reduced with a given clause while the *resolve* function specifies the effect of the reduction on the constraint store.

Definition 3.1 [Resolve and try (concrete)]. The *concrete resolve function*, $resolve_C : Atom \times Con_C \times Clause \rightarrow Con_C$, is defined by

$$resolve_C(A, \theta, C) = (A = H \wedge Ask \wedge Tell \wedge \theta)$$

and the *concrete try function*, $try_C : Atom \times Con_C \times Clause \rightarrow \{fail, success, delay\}$, by

$$try_C(A, \theta, C) = \begin{cases} fail & \text{if } resolve_C(A, \theta, C) = fail \\ success & \text{if } resolve_C(A, \theta, C) \neq fail \\ & \quad \text{and } \theta \leq \bar{\exists}_A(A = H \wedge Ask) \\ delay & \text{otherwise} \end{cases}$$

where $C = H : - \ Ask : Tell \mid B$.

A constraint has an associated "stuck" relationship that holds for an atom if under the constraint the atom cannot be successfully reduced with any clause, but does delay for some clause. In general, $stuck \subseteq Atom \times Con$.

Definition 3.2 [Stuck atom (concrete)]. Let $P \in Prog$ and $\theta \in Con_C$. Atom A is *stuck* in θ (for P), written $stuck_C^P(A, \theta)$, if:

1. $\exists C \ll_V P.\ try_C(A, \theta, C) = delay$; and
2. $\forall C \ll_V P.\ try_C(A, \theta, C) \neq success$ where $V = vars(\langle A; \theta \rangle)$.

Typically, implementations of the operational semantics use a particular process scheduling policy, such as selection of the leftmost nonstuck process. We model a scheduling policy as an arbitrary choice of atoms from the state's goal that satisfies a suitable scheduling relation.

Definition 3.3 (Scheduling relation). Let $P \in Prog$. A *scheduling relation*, $sched^P \subseteq State \times Goal$, is defined in terms of a corresponding relation $stuck^P \subseteq Atom \times Con$ by: $sched^P(s, B)$ if:

1. $B \subseteq goal(s)$;
2. $A \in B \Rightarrow \neg\ stuck^P(A, con(s))$; and
3. $\exists A \in goal(s).\neg\ stuck^P(A, con(s)) \Rightarrow B \neq \emptyset$.

In the following, we will often be interested in two specific concrete scheduling relations: the *standard* scheduling relation, denoted $sched_C^P$, which is defined in terms of the relation $stuck_C^P$ defined in Definition 2; and the *confluent* scheduling relation defined in the next section. Reduction is defined in terms of a set of selected atoms.

Definition 3.4 (Concrete resolvent). Let $P \in Prog$ and $s, s' \in State_C$. The state s' is a *concrete resolvent* of s with atom A and defining clause $C = H :- Ask : Tell \mid B$ if $A \in goal(s)$ and $s' = \langle (goal(s) \backslash \{A\}) \cup B;\ resolve_C(A, con(s), C) \rangle$. In this case, A is said to be the *selected atom* and C the *corresponding clause.*

The standard reduction relation for P, $reduce_C^P \subseteq State_C \times State_C \times Atom$, is defined by $reduce_C^P(s, s', A)$ if s' is a concrete resolvent of s with A and $C \ll_s P$ such that $try_C(A, con(s), C) = success$.

In the following, the superscript will be omitted from $stuck_C^P$, $sched_C^P$, and $reduce_C^P$ when clear from the context. The operational semantics is defined as a *transition system*, which is a graph that has nodes labeled by states. The initial state is a "source node" and edges correspond to *reductions* between the states. Thus, reduction sequences correspond to paths in the graph starting from the source. Different transition systems for the same initial state and program result from different scheduling rules.

Definition 3.5 (Transition system). Let $P \in Prog$, $State$ be a set of states, $s \in State$, and let $reduce \subseteq State \times State \times Atom$ and $sched \subseteq State \times Goal$ be a *reduction relation* and a *scheduling relation*, respectively. A *transition system* $\mathcal{G}$ for P and s is a graph with each node n labeled by a state, denoted by $state(n)$, and a set

of selected atoms, denoted by $sel(n)$ such that:

1. every node in $\mathcal{G}$ is reachable from a distinguished node called the source, which is labeled with state s;
2. for all nodes n, $sched(state(n), sel(n))$; and
3. for each node n and each $A \in sel(n)$, if $reduce(state(n), s, A)$ for some state s, then there is a node n' with $state(n') = s$ and an arc from n to n'.

A transition system has a corresponding relation $stuck \subseteq Atom \times Con$ inherited from *sched*. Note that *reduce* determines which clauses can be used, while *sched* determines which goals are scheduled. The standard operational semantics of a program is given by a *standard transition system*, which is a transition system constructed from concrete states and the standard scheduling and reduction relations. Figures 1 and 2 are examples of standard transition systems for the initial state and program of Example 2.1.

The execution sequences of a program are modeled by its *derivations*.

Definition 3.6 (Derivation). Let $\mathcal{G}$ be a transition system on states *State* for $s_1 \in$ *State* and $P \in$ *Prog*. A *derivation* in $\mathcal{G}$ is a (possibly infinite) maximal sequence of states $s_1 \rightarrow s_2 \rightarrow \cdots$ such that there is a path in $\mathcal{G}$ with s_i the state label of the ith node in the path. A *partial derivation* need not be maximal. When we also wish to indicate the selected atom and clause at each stage, we write the derivation as $s_1 \stackrel{A_1;C_1}{\longrightarrow} s_2 \stackrel{A_2;C_2}{\longrightarrow} \cdots$.

For example, the transition system shown in Figure 2 has a single derivation:

$$\langle p(x), q(y); x = y\rangle \rightarrow \langle q(y); x = b \wedge y = b\rangle \rightarrow \langle true; x = b \wedge y = b\rangle.$$

The declarative behavior of a program and state is given by its successful and failed derivations.

Definition 3.7 (Successful–suspended–failed state). Let P be a program and $s \in$ *State* a state. For a given *stuck*, we say that:

1. s is a *success* state for P if $goal(s) = true$;
2. s is a *suspended* state for P if $goal(s) \neq true$ and for all $A \in goal(s)$, $stuck(A, con(s))$; and
3. otherwise, s is a *failed* state for P.

Definition 3.8 (Successful–suspended–failed derivation, answer). Let $\mathcal{G}$ be a transition system on *State* for $s \in$ *State* and $P \in$ *Prog* with corresponding *stuck*. A (finite) derivation which ends in a state s' is:

1. *successful* if s' is a success state for P;
2. *suspended* if s' is a suspended state for P given *stuck*; and
3. is *failed* otherwise.

An *answer* of a transition system $\mathcal{G}$ for state s is a constraint $\exists_s con(s')$ where s' is the last state of a successful derivation for s in $\mathcal{G}$.

For instance, the single derivation transition system shown in Figure 2 is a successful derivation with answer $x = b \wedge y = b$.

4. DIFFERENT TYPES OF SUSPENSION

Our primary purpose is to illustrate the use of confluent semantics as a basis for analyses of reactive aspects of program behavior, particularly the analysis of various types of suspension. Roughly speaking, these tell us whether a particular process requires input from other processes to continue. The most obvious kind of suspension occurs whenever a process reaches a state in which all atoms are stuck.

Definition 4.1 (Suspends). A transition system *suspends* if it has a derivation that suspends. A state s *leads to suspension* for P if there is a standard transition system for P and s that suspends.

For instance, the initial state and program from Example 2.1 lead to suspension, as the (standard) transition system shown in Figure 2 suspends.

The above definition of suspension specifies the global behavior of a system formalizing the concept generally assumed in the context of concurrent (constraint) logic languages (e.g., [3, 4, 6, 24, 26]). However, we are often interested in the local behavior of a single process within a system. We introduce the following notion of local suspension, which is similar to the notion of deadlock of an agent in CSP [11]. As we have seen from Example 2.4, a notion of fairness is required to formalize local suspension.

Definition 4.2 (Fair derivation). An infinite derivation $s_1 \xrightarrow{A_1;C_1} s_2 \xrightarrow{A_2;C_2} \cdots$ is *fair* if for every $i \in I\!N$, $A \in goal(s_i) \wedge \neg\, stuck(A, con(s_i)) \Rightarrow (\exists j \geq i.A = A_j)$. A transition system is *fair* if all of its infinite derivations are fair.

For example, the transition system in Figure 5 is fair.

Definition 4.3 (Local suspension). Let $\mathcal{G}$ be a transition system on *State* for program P and $s \in State$. A derivation $s_1 \rightarrow s_2 \rightarrow \cdots$ of $\mathcal{G}$ *locally suspends* if it is not failed and there is some state s_i such that for some $A \in goal(s_i)$, $stuck(A, con(s_j))$ holds for all $j \geq i$. A transition system *locally suspends* if it has a derivation that locally suspends. A concrete state s *leads to local suspension* for program P if there is a fair standard transition system for P and s that locally suspends.

For example, the initial state and program from Example 2.3 locally suspend, while those from Example 2.4 do not. Clearly, suspension implies local suspension.

We note that absence of suspension, global and local, is not preserved under state composition.

Definition 4.4 (State composition). The *composition* of $s, s' \in State_C$ is the state $s \parallel s' = \langle goal(s) \cup g; con(s) \wedge con(s') \wedge g = goal(s') \rangle$, where $g \ll_{\langle s,s' \rangle} \{goal(s')\}$.

Consider, for instance, the states $\langle p(x); true \rangle$ and $\langle q(x); true \rangle$ with the program from Example 2.1. Neither leads to suspension, while their composition does. We now introduce a confluent operational semantics which is the basis for the suspension analyses proposed in the following sections. Interestingly enough, the absence of suspension, global and local, is preserved under composition with respect to this semantics.

5. A CONFLUENT SEMANTICS

In this section, we present a confluent semantics which approximates the standard operational semantics. The basic idea is to separate synchronization from nondeterminism by interpreting synchronization at the procedure level. This is achieved by: (1) scheduling an atom only if it can be scheduled in the standard semantics and cannot become stuck under further instantiation, and (2) synchronizing a clause not by its own synchronization condition, but rather by a combined condition consisting of a disjunction of the conditions from the clauses of the given procedure. Hence, if an atom can reduce with any clause, then it can reduce with all clauses which are consistent with the current constraint.

Definition 5.1 (Confluent transition system). Let $P \in Prog$. The *confluent reduction relation*, $reduce_{\mathcal{CF}} \subseteq State_{\mathcal{C}} \times State_{\mathcal{C}} \times Atom$, is defined by $reduce_{\mathcal{CF}}(s, s', A)$ if s' is a concrete resolvent of s with A and $C \ll_s P$ such that $try_{\mathcal{C}}(A, con(s), C) \neq fail$. The *confluent scheduling relation*, $sched^P_{\mathcal{CF}}$, is the concrete scheduling relation determined by

$$stuck^P_{\mathcal{CF}}(A, \theta) = \exists \theta' \leq \theta.\ stuck^P_{\mathcal{C}}(A, \theta').$$

A *confluent transition system* is a transition system constructed from $State_{\mathcal{C}}$ with the scheduling relation $sched^P_{\mathcal{CF}}$ and the confluent reduction relation $reduce_{\mathcal{CF}}$.

The following theorem justifies our use of the term "confluent." It states that independence of scheduling holds for finite derivations and for infinite derivations in the case that the transition systems are fair, in the sense that different confluent transitions systems have "isomorphic" derivations.

Definition 5.2 (Isomorphic derivations). Let $\mathcal{G}$ and $\mathcal{G}'$ be transition systems for s and P. Let $\mathcal{D}$ and $\mathcal{D}'$ be derivations of $\mathcal{G}$ and $\mathcal{G}'$, respectively, with $\mathcal{D} = s_1 \overset{A_1;C_1}{\longrightarrow} s_2 \overset{A_2;C_2}{\longrightarrow} \cdots \overset{A_i;C_i}{\longrightarrow} s_{i+1} \cdots$ where the s_i are indexed by I, a possibly infinite initial subsequence of the positive integers. We say that $\mathcal{D}$ is *isomorphic* to $\mathcal{D}'$ if there exist a renaming $\mathcal{D}''$ of $\mathcal{D}'$ and a bijection $f : I \to I$ such that $\mathcal{D}'' = s_1 \overset{A_{f(1)};C_{f(1)}}{\longrightarrow} s'_2 \overset{A_{f(2)};C_{f(2)}}{\longrightarrow} \cdots \overset{A_{f(j)};C_{f(j)}}{\longrightarrow} s'_{j+1} \cdots$. We say that $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic* if their derivations are isomorphic.

It is straightforward to observe that the transition systems illustrated in Figures 1 and 2 are not isomorphic.

Theorem 5.1. Let $\mathcal{G}, \mathcal{G}'$ be confluent transition systems for s and P, and let $\mathcal{D}$ be a nonfailing derivation of $\mathcal{G}$. Then,

(1) *if $\mathcal{D}$ is finite, then there is a finite nonfailing derivation of $\mathcal{G}'$ isomorphic to $\mathcal{D}$; and*
(2) *if $\mathcal{D}$ is infinite and $\mathcal{G}$ and $\mathcal{G}'$ are fair, then there is an infinite derivation of $\mathcal{G}'$ isomorphic to $\mathcal{D}$.*

This means that if $\mathcal{G}$ and $\mathcal{G}'$ are confluent transition systems for a given program and initial state, then they will have the same answers, and each will suspend if and only if the other does. This is the main technical contribution of the paper. It is easy

to construct an example showing that if the fairness requirement is dropped, then the theorem no longer holds. The proof of the theorem is given in the Appendix.

The following results justify the use of the confluent semantics as a basis for analyzing various suspension properties. They imply that suspension analyses can be based on a single scheduling rule and inspection of the resulting confluent transition system.

Theorem 5.2. If $s \in State_C$ and $P \in Prog$ lead to suspension, then any confluent transition system for s and program P suspends.

Corollary 5.1. A state s does not lead to suspension for P if there exists a confluent transition system for s and P that does not suspend.

Theorem 5.3. If $s \in State_C$ and $P \in Prog$ lead to local suspension, then any fair confluent transition system for s and P locally suspends.

Corollary 5.2. A state s does not lead to local suspension for program P if there exists a fair confluent transition system for s and P that does not locally suspend.

As noted above, the absence of local and global suspension in the confluent semantics is closed under state composition.

Proposition 5.1. Let $s, s' \in State_C$. If there exist (fair) confluent transition systems for P with s and with s' which do not (locally) suspend, then there exists a (fair) confluent transition system for the composed state $s \parallel s'$ which does not (locally) suspend with P.

For many programs, P, the confluent and standard semantics are equivalent in the sense that for all $s \in State_C$, $\mathcal{G}$ is a standard transition system for s and P if and only if it is a confluent transition system for s and P. These are programs that are not affected by lifting the interpretation of synchronization from the clause level to the procedure level. In particular, P-Prolog [28] and P-Prolog$_x$ [26] programs are specifically defined in this way.

Definition 5.3 (Confluent program). A program P is *confluent* if for all atoms A, constraints θ, and (distinct) clauses $C \ll_{\langle A;\theta\rangle} P$ and $C' \ll_{\langle A;\theta\rangle} P$,

$$try_C(A, \theta, C) = success \Rightarrow try_C(A, \theta, C') \neq delay.$$

In particular, programs for which the ask parts of the guards are mutually exclusive are confluent because they are deterministic in the strong sense that, if an atom succeeds with some clause, it will fail with all other clauses.

Proposition 5.2. The confluent and standard transition semantics are equivalent for confluent programs.

Consequently, independence of the scheduling rule holds for confluent programs using the *standard semantics*. Examples of programs which are not confluent and for which the standard and confluent transition systems differ are the programs from Examples 2.1 and 2.4.

6. SUSPENSION ANALYSES

In this section, we sketch analyses for suspension and local suspension. We couch these analyses as abstract interpretations [8] of the confluent semantics in which constraints are replaced by constraint descriptions. Correctness is argued by providing an approximation relation, $\propto$, which relates the descriptions to the objects they describe.

Definition 6.1. A *description* $\langle D, \gamma, E\rangle$ consists of a *description domain* (a poset) D, a *data domain* (a poset) E, and a monotonic *concretization function* $\gamma : D \to \wp E$. When $E = Con$, the description is called a *constraint description.*

We say that d γ-approximates e, written $d \propto_\gamma e$, iff $e \in \gamma(d)$. The approximation relation is lifted to functions as follows. Let $\langle D_1, \gamma_1, E_1\rangle$ and $\langle D_2, \gamma_2, E_2\rangle$ be descriptions, and let $F : D_1 \to D_2$ and $F' : E_1 \to E_2$ be functions. Then $F \propto F'$ iff $\forall d \in D_1.\ \forall e \in E_1.\ d \propto_{\gamma_1} e \Rightarrow F(d) \propto_{\gamma_2} F'(e)$. When clear from the context, we say that d approximates e and write $d \propto e$.

In a suspension (or local suspension) analysis, one is interested in knowing which atoms in a state are "possibly" stuck. This requires that the constraint description keep information about how variables in a state will be affected if other variables in that state become more instantiated after a reduction. In [4], we have given several such domains. Below, we formalize one such domain based on *depth k* descriptions [25] of states, which are simple yet suffice to illustrate the analyses of nontrivial programs. Additionally, we sketch the use of a second, more complicated domain via an example.

An analysis based on our method attempts to construct an "abstract transition system" that is (local) suspension-free. One way to ensure termination of such an analysis is to guarantee that the analyses will be based on abstract transition systems which contain a finite number of states. The number of constraints that occur in such states can be restricted by using finite constraint descriptions or by applying widening operations [8]. As illustrated in [4], it is also necessary to restrict the number of atoms which might occur in states. This is due to the dynamic nature of processes in concurrent constraint languages. A technique termed *star abstraction*, which guarantees termination of analyses, is described in [4]. For simplicity of presentation, we focus here on analyses which are based on a more simple state description which is induced from $State_C$ by leaving the goal component as it is, and abstracting the constraint component of the state.

Definition 6.2 (Abstract states). Let $Con_\mathcal{A}$ be a constraint description. Define the *induced (abstract) states*, $State_\mathcal{A}$, to be $Goal \times Con_\mathcal{A}$. Let $t \in State_\mathcal{A}$, $s \in State$ and $\rho \in Ren$. Define $t \propto_\rho s$ iff: (1) $goal(t) = \rho(goal(s))$, and (2) $con(t) \propto \rho(\exists_{goal(s)} con(s))$. We say that t approximates s, written $t \propto s$, iff $\exists\, \rho \in Ren.t \propto_\rho s$.

Reduction of abstract states is essentially the same as for concrete (confluent) reduction. To ensure correctness of the analysis, we require that the abstract reduction relation $reduce^P_\mathcal{A}$ approximates the confluent reduction relation $reduce^P_{\mathcal{CF}}$, namely, that $reduce^P_\mathcal{A} \subseteq State_\mathcal{A} \times State_\mathcal{A} \times Atom$ satisfies the following for all $A \in Atom, s, s' \in State_C$ and $t \in State_\mathcal{A}$:

$$t \propto_\rho s \ \&\ reduce^P_{\mathcal{CF}}(s, s', A) \Rightarrow \exists t' \in State_\mathcal{A}.reduce^P_\mathcal{A}(t, t', \rho(A))\ \&\ t' \propto s'.$$

This ensures that, if there is a reduction in the confluent semantics, there will be a corresponding reduction in the approximating abstract semantics. We must also define what it means for an atom to be stuck in an abstract constraint. As we are interested in detecting possible suspension and local suspension, we should err on the side of being stuck. Thus, an atom is stuck in the abstract constraint whenever the atom is stuck in some concrete constraint described by the abstract constraint. That is, atom A is stuck in the abstract constraint θ for program P, written $stuck^P_{\mathcal{A}}(A, \theta)$, if $\exists E \in Con_{\mathcal{C}}.\ \theta \propto E\ \&\ stuck^P_{\mathcal{CF}}(A, E)$. The definitions for scheduling relation, suspended state, derivation, etc., are as for the concrete states, except that abstract stuckness replaces concrete stuckness. An abstract transition system is a transition system with states $State_{\mathcal{A}}$, a scheduling relation $sched_{\mathcal{A}}$ defined in terms of $stuck_{\mathcal{A}}$, and an abstract reduction relation $reduce_{\mathcal{A}}$.

We analyze a state s and program P for possible suspension by constructing a single abstract transition system for s and P. If this system contains no suspended derivations, then s and P definitely do not lead to suspension (for any scheduling rule) with respect to the standard transition system; otherwise, they may. Similarly, we analyze s and P for possible local suspension by constructing a single fair abstract transition system for s and P. If this system contains no locally suspended states, then s and P definitely do not locally suspend (for any fair scheduling rule); otherwise, they may. Correctness of these analyses is a consequence of the following theorem.

Theorem 6.1. Let $\mathcal{G}$ be an abstract transition system for a state s and program P. If $\mathcal{G}$ is fair and does not lead to (local) suspension, then any standard transition system for s and P does not lead to (local) suspension.

Observe that an unfair derivation can starve a process even though it is not stuck. Hence, our notion of a state *leading to local suspension* restricts attention to fair derivations from that state. However, when constructing an abstract transition system during program analysis, fairness has another important implication. An unfair derivation can fail to introduce a process that would remain stuck forever. Thus, the abstract transition system constructed by the analysis must be fair. For example, the state $s = \langle clock(x), p; x = tick\rangle$ with the program from Example 2.3 extended by the clause

```
p :- true : true | q(x).
```

has an unfair derivation that avoids local suspension by always scheduling `clock` (`p` is not stuck). An abstract transition system based on this scheduling would not constitute a legitimate analysis of the program, since s leads to local suspension by scheduling `p`, introducing `q(x)`, which remains forever stuck.

In the following, we assume for simplicity that concrete constraints are constructed from syntactic equations over the Herbrand universe. We denote by *Eqn* the set of possibly existentially quantified conjunctions of syntactic equations. The idea is to approximate a constraint by a constraint in which the depth of the terms is bounded by k.

Definition 6.3 (Depth k abstraction). We say that an element $E = \exists Y.\theta$ of *Eqn* is *solved* if θ has the form $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$ such that each x_i is a distinct variable not occurring in any of the terms t_i and each $y \in Y$ occurs in some t_j. We say that an equation $\exists Y.\theta$ in solved form is of *depth* k if $\forall(x = t) \in \theta.depth(t) \leq k$

where $depth(t)$ is the depth of term t. We denote the set of depth k equations by Eqn_k.

An equation $\exists Y.\theta \in Eqn_k$ is intended to describe any equation $E \in Eqn$ that can be obtained by further instantiating the variables in Y. More precisely, $\exists Y.\theta$ approximates equation E, written $\exists Y.\theta \propto E$, if

$$\exists E' \in Eqn.E \Leftrightarrow \exists_Y.(\theta \wedge \bar{\exists}_Y.E').$$

The correctness of the abstract reduction function induced by this domain is straightforward. A similar proof has been given in [4].

The confluent transition system for Example 5 given in Figure 5 is also a depth 2 local-suspension analysis for the same program. This illustrates that our technique is adequate to analyze bidirectional communication by using a domain as simple as depth 2.

7. TYPES AND SUSPENSION ANALYSES

In the above discussion, we have illustrated the utility of abstracting a confluent semantics in the analysis of reactive program properties. The analyses presented so far are simplistic in the notion of data abstraction assumed, namely, depth k abstraction. This approach is particularly restrictive for the class of languages with atomic publication. In this case, it is essential to have a stronger notion of data abstraction in order to provide a suitable definition of confluent stuckness. In this section, we demonstrate how the technique of abstracting a confluent semantics can incorporate more substantial notions of data abstraction. We choose a simple notion of type information and illustrate two example analyses. The first is a simple program involving two producers, a merge predicate and a single consumer. The merge predicate is an example of a predicate with two inputs that consumes from only one of these in each reduction step. The analysis of this type of behavior typically requires more powerful constraint descriptions. The first example illustrates that our technique is also adequate to analyze such programs. Note that this program is not confluent.

Example 7.1. The following program P defines two producers, *pa* and *pb*, which, respectively, produce a stream of a's and b's. P also defines a process m, which nondeterministically merges its two input streams, produced by *pa* and *pb*, into an output stream consumed by a consumer, c, which consumes a stream of a's and b's.

```
(1) pa(x) :- true : x = [a| x1] | pa(x1).
(2) pb(x) :- true : x = [b| x1] | pb(x1).

(3) c(x) :- x =[a| x1] : true | c(x1).
(4) c(x) :- x =[b| x1] : true | c(x1).

(5) m(x,y,z) :- x = [u| x1] : z = [u| z1] | m(x1,y,z1).
(6) m(x,y,z) :- y = [u| y1] : z = [u| z1] | m(x,y1,z1).
```

We analyze this program for the initial state

$$s = \langle pa(x_1), pb(x_2), m(x_3, x_4, x_5), c(x_6); (x_1 = x_3 \wedge x_2 = x_4 \wedge x_5 = x_6)\rangle,$$

in which the streams produced by *pa* and *pb* are merged and fed into c. A transition system for P and s is illustrated in Figure 6. For this example, we use a more

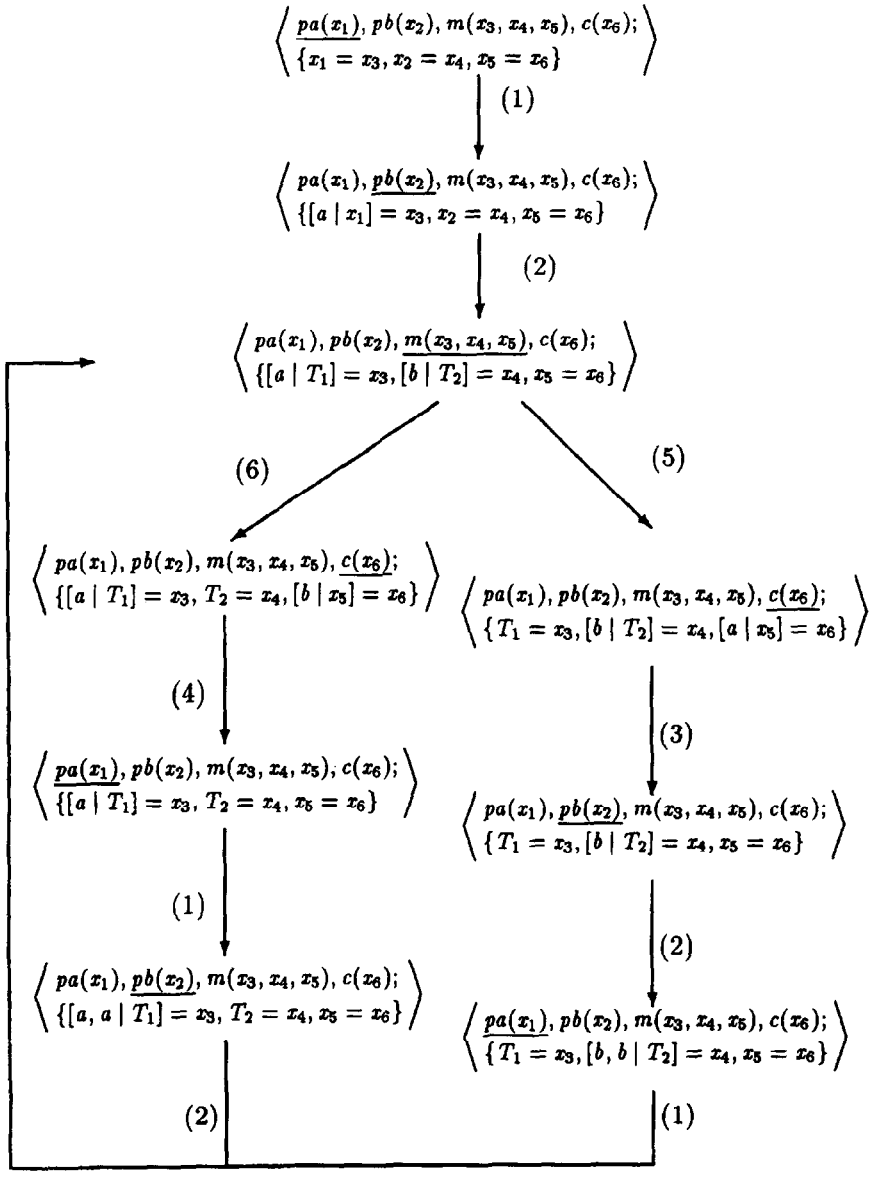

FIGURE 6. Abstract transition system for merge.

sophisticated abstract domain based on regular trees [13, 19]. The type variables T_1 and T_2 define sets of finite terms by the grammar

$$T_1 ::= x_1 \mid [a \mid T_1]$$
$$T_2 ::= x_2 \mid [b \mid T_2]$$

The concrete constraints approximated by an abstract constraint involving type variables are obtained from the abstract constraint by replacing each occurrence of a type variable by one of the terms of that type (not necessarily replacing all

occurrences of a given type variable uniformly by the same member). Thus, $[a, a \mid T_1] = x_3 \propto [a, a \mid t_1] = x_3$, for all $t_1 \in T_1$. For instance, $[a, a \mid T_1] = x_3 \propto [a, a, a \mid x_1] = x_3$ since $[a \mid x_1] \in T_1$. Note that under this interpretation, concrete constraints are also abstract constraints. For instance, $[a \mid x_1] = x_3 \propto [a \mid x_1] = x_3$. The transition system shown in Figure 6 uses widening operations [8] based on the observation that for any concrete constraint c, $[a \mid x_1] = x_3 \propto c \Rightarrow [a \mid T_1] = x_3 \propto c$ and $[a, a \mid T_1] = x_3 \propto c \Rightarrow [a \mid T_1] = x_3 \propto c$.

Recursive types are helpful here because, each time we reduce the merge predicate, we must reduce both producers to ensure fairness. Doing so with the depth k abstraction would quickly lose track of the relationship between the producers' variables and the merge's variables, yielding stuck atoms. As the transition system does not contain a suspended state, the analysis shows that the original program and state will not suspend. Furthermore, the abstract transition is fair, and it has no locally suspending derivations. Therefore, the original state and program do not locally suspend.

The second example in this section is an analysis for a version of the Dining Philosophers problem taken from [26].

Example 7.2. The Dining Philosophers is a classical example which illustrates the problem of mutual exclusion. The philosophers are sitting around a round table, and there is one fork shared by each pair of adjacent philosophers. Each philosopher goes through cycles of eating and thinking. The problem is to provide an algorithm which guarantees that the philosophers will not deadlock, and that no philosopher will starve. Shapiro [26] presents a program in which the implementation of mutual exclusion is facilitated by the atomic publication mechanism of the language. Let us see how we can prove that the given program does not suspend.

```
(1) phil(Id,Ls,Rs) :- true : (Ls=[Id|Ls'],
                               Rs=[Id|Rs']) | phil(Id,Ls',Rs').
(2) phil(Id,Ls,Rs) :- Ls=[ _ |Ls'] : true | phil(Id,Ls',Rs).
(3) phil(Id,Ls,Rs) :- Rs=[ _ |Rs'] : true | phil(Id,Ls,Rs').
```

The program specifies the behavior of an individual philosopher. Each philosopher has a unique identifier Id, and is connected to his left and right neighbor by streams Ls and Rs which specify the activity of the corresponding left and right forks. Let us consider a dinner eaten by three philosophers. Hence, we want to analyze the initial state

$$s = \langle phil(1, F_1, F_1'), phil(2, F_2, F_2'), phil(3, F_3, F_3');$$
$$F_1' = F_2 \wedge F_2' = F_3 \wedge F_3' = F_1 \rangle.$$

A philosopher tries to grab both forks, excluding other philosophers from grabbing them. Mutual exclusion is obtained by unifying the head of the stream with the unique Id of the philosopher. Each stream F_i is instantiated to the sequence of the identifiers of those philosophers which succeeded to use the ith fork.

In a simplistic approach based on depth k abstraction, each of the processes in s must be considered (abstract) stuck. To see this, consider $phil(1, F_1, F_1')$ and an instance where F_1 is bound to a term which is not a list and F_1' is not bound.

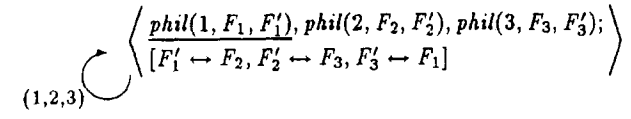

FIGURE 7. Suspension analysis for dining philosophers.

For analyses of this kind, greater precision can be obtained by utilizing more information about the eventual bindings of variables. Suspension analyses based on a confluent semantics and type analysis can be formalized by introducing a suitable notion of confluent stuckness into the definition of confluent transition system. Given a type environment Γ for a program P, we can define a corresponding notion of confluent stuckness:

$$stuck^{P,\Gamma}_{\mathcal{CF}}(A,\theta) = \exists \theta' \leq_\Gamma \theta.\ stuck^P_{\mathcal{C}}(A,\theta').$$

Intuitively, Γ specifies that each variable can be bound only to terms of specific types. In general, when we write $\theta' \leq_\Gamma \theta$, we mean that $\theta' \Rightarrow \theta$ and that $\theta' \wedge \Gamma$ is consistent.

Now, reconsidering the example of the Dining Philosophers, let us assume a simple type analysis for logic programs which guarantees that F_1 and F_1' in $phil(1, F_1, F_1')$ are always bound to list structures (see, e.g., [13, 27]). In this case, the suspension and local suspension analyses become trivial within our framework. In fact, none of the processes is ever stuck. For each process, the first clause can apply if the process is not idle, or otherwise the second or third clause will apply. The domain which we consider for these analyses is the extractor sharing description [4]. The idea of extractor sharing descriptions is to use functions which extract some "distinguished" subterms from a term.

Example 7.3 [4]. The following is a semi-linear extractor which extracts the tail of a list (or stream):

$$tail(t) = \begin{cases} tail(t') & \text{if } t = [h \mid t'] \\ \{t\} & \text{otherwise.} \end{cases}$$

So, $tail([y, b \mid x]) = \{x\}$ and $ground_{tail}([a, b \mid nil])$ is true.

By using this description, we obtain the graph of Figure 7 for the analysis of suspension and that of Figure 8 for the analysis of local suspension. An abstract equation such as $[x \leftrightarrow x']$ means that the variables x and x' can be bound to lists which possibly contain shared variables in their tails.

8. EVENTUAL PUBLICATION

The above discussion addresses languages with atomic publication. Languages using eventual publication yield stronger results. The semantics of these languages require us to consider a different definition of the concrete try function (see Definition 1): $try_{\mathcal{C}} : Atom \times Con_{\mathcal{C}} \times Clause \to \{fail, success, delay\}$ is now defined by

$$try_{\mathcal{C}}(A,\theta,C) = \begin{cases} fail & \text{if } \theta = fail \\ success & \text{if } \theta \neq fail \text{ and } \theta \leq \bar{\exists}_A(A = H \wedge Ask) \\ delay & \text{otherwise} \end{cases}$$

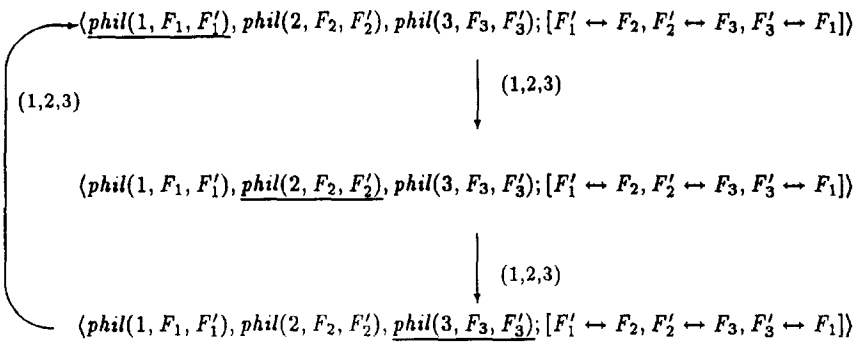

FIGURE 8. Local suspension analysis for Dining Philosophers.

where $C = H : -Ask : Tell \mid B$. The following results follow when this definition of try_C replaces that of Definition 1.

Proposition 8.1. Let $P \in Prog, A \in Atom$, and $\theta \in Con_C$. Then $stuck_C^P(A, \theta) \Leftrightarrow stuck_{C\mathcal{F}}^P(A, \theta)$.

From this equivalence, it follows easily that any concrete derivation is also a confluent derivation.

Proposition 8.2. Let $P \in Prog$ and $s \in State_C$, and let $\mathcal{G}$ be a concrete transition system for P and s. There exists a confluent transition system $\mathcal{G}'$ for P and s that contains $\mathcal{G}$ as a subgraph.

Combining this result with Theorem 5.1, we obtain the following.

Corollary 8.1. Let $P \in Prog$ and $s \in State_C$. Let $\mathcal{G}$ be a fair concrete transition system, and $\mathcal{G}'$ a fair confluent transition system for P and s. For any nonfailed derivation $\mathcal{D}$ of $\mathcal{G}$, there exists a derivation $\mathcal{D}'$ of $\mathcal{G}'$ isomorphic to $\mathcal{D}$.

Theorems 5.2 and 5.3, along with their corollaries, now follow.

9. RELATED WORK

The main contribution of this paper is an approximate confluent semantics for concurrent constraint logic programs which forms the basis for their efficient and accurate analysis. To our knowledge, this idea of basing the analyses of a concurrent language on an approximating confluent semantics is new. Our specific technical contributions are twofold. First, we show that our semantics is confluent, and that it approximates the standard semantics. Second is the definition and analysis for local suspension, which has not been previously studied in the context of concurrent logic languages.

Independence of scheduling in the confluent semantics for finite computations generalizes in a sense the standard result for logic programming, i.e., Theorem 9.2 in [15], Theorem 4 in [29] for moded equational programs, and Theorem 3.7 in [23] for determinate concurrent constraint programs. In particular, this result could

be applied to definite programs with delay [21]. Independence of scheduling in the confluent semantics for fair infinite computations generalizes Theorem 6.5 [14] for logic programming.

Our definition of suspension is the one generally assumed in the context of concurrent (constraint) logic languages [24, 26]. The definition of local suspension and its corresponding analysis are novel. Local suspension is similar to the notion of deadlock of an agent in CSP [11]. The suspension and local suspension analyses we give have some similarities to analyses of CSP, for example, see [1, 2], and to the work of Peng and Purushothaman [22] as our analyses essentially construct a graph of possible states of the processes and arcs which link possible reductions between those states. However, there are significant differences reflecting the different underlying models of concurrency. In particular, we must handle asynchronous communication and dynamic creation and deletion of processes and communication channels.

Codognet et al. [6] and Codish et al. [4] have also investigated the analysis of concurrent logic languages, and in particular, analyses for the detection of possible suspension. The paper [6] gives two analysis algorithms based on the abstraction of the AND–OR tree of a program and goal. The first algorithm requires computing all possible interleavings. As the authors note, this is not very practical. The second algorithm overcomes this by performing local reexecution of atoms in clause bodies. This is more efficient than computing all possible interleavings, but is inherently less precise. This approach is orthogonal to our use of confluence. Another difference is that their analysis framework is based on the AND–OR tree semantics, while ours is substantially simpler because we directly abstract the transition system semantics.

The results of this paper are closely related to those presented in [4] where analyses are also based on abstracting a transition system semantics. It also defines restrictions on analyses which ensure independence of scheduling for finite computations, allowing efficient implementation. The present work is an improvement for two reasons: first, because it considers the confluence of nonterminating computations which enables the analyses of local suspension; and second, because it simplifies the formal justification of analyses which can now be seen directly as abstractions of the confluent semantics. Other research on the analysis of concurrent constraint logic programs includes that of Horiuchi [12]. This work differs from the approach taken here as it is based on a denotational semantics and because all interleavings must be considered.

Recently, there has also been work in the analysis of concurrent constraint programs (ccp). Unlike our context of concurrent logic languages, ccp is a programming language with eventual publication [23]. Falaschi et al. [9] describe an approach based on local reexecution, formalized in terms of a denotational semantics using closure operators. Subsequently, our approach involving the approximate confluent semantics has also proved useful in the analysis of ccp programs. The confluent semantics given here is simplified in the ccp context of eventual publication. Hence, confluence can be expressed in terms of a source-to-source transformation from concurrent constraint programs to concurrent constraint programs which are confluent under the usual operational semantics. Such a transformation has been given independently by Falaschi et al. [10] and Zaffanella et al. [30]. Both of these papers abstract denotational semantics for these languages.

Zaffanella et al. [30] investigate the analysis of suspension-free concurrent constraint programs. In particular, they show two possible transformations on

suspension-free ccp programs which allow the application of standard constraint logic programs data flow analysis techniques to the transformed programs. Zaffanella [31] studies a class of program properties for which the approximation of the choice operator becomes less problematic. In [10], Falaschi et al. combine the reexecution approach of [9] with the confluent approach. This provides a compositional semantics for a subset of ccp in which not all interleavings need to be considered. In a recent paper [7], Codognet et al. apply a denotational semantics for ccp programs with angelic nondeterminism, and discuss the use of this semantics as a basis for program analysis. This work differs from ours as it does not consider the standard "indeterministic" choice operator. It is interesting to note that a new operator (Guarded Constructive Disjunction) which is deterministic, and hence clearly confluent, is introduced by [7] for the purpose of analysis.

The analysis of sequential logic languages with flexible scheduling has been investigated by Marriott et al. [16]. Flexible scheduling means that computation generally proceeds left to right, but some calls are dynamically "delayed" until their arguments are sufficiently instantiated. These languages differ from the setting considered here as they have a default left-to-right scheduling; hence, they cannot ensure that an analysis will be correct for any scheduling. The interaction between nondeterminism and synchronization is somewhat simpler in this kind of sequential language.

The recent work of Marriott and Oderski [17] presents a confluent calculus for ccp. Their semantics is precise in the sense that the observables of the semantics correspond to those of standard ccp. However, the semantics itself is significantly more complex than ours since the calculus has to keep track of those branches of the search tree which have not been completely evaluated.

10. CONCLUSION

We have introduced a confluent semantics for concurrent constraint logic languages which provides a basis for their efficient and accurate analysis. We have shown the usefulness of this approach by using it to develop analyses for suspension and local suspension.

Another potential application that we have not discussed is to optimize implementations by applying a scheduling rule for which a program is shown not to locally suspend. In this case, speed-ups may be obtained as there is no need to test for synchronization. Moreover, the need for intervention by a runtime scheduler to ensure fairness is eliminated as the abstract transition system is fair. Thus, these systems may provide a basis for increasing the granularity of parallelism in the implementation.

As future work, we mention the study of appropriate abstract scheduling rules to prove local suspension freeness, and also the use of types to achieve stronger suspension analyses.

APPENDIX
PROOF OF THEOREMS

Most of the following proofs discuss derivations and require identification of atoms within derivations. For convenience in identifying atoms, we modify the original program and the initial state so that all atoms contain at least one variable. This

is accomplished by adding a single-occurrence variable as an extra argument to each predicate that has an atom with no variables in a clause body or in the initial state. It does not significantly alter the reduction semantics because a state is based on a multiset of goals rather than a set. For each of the semantics, there is a bijection between the derivations of the original and the modified program such that the paired derivations differ only by the addition of the dummy arguments. The constraints in corresponding derivations are identical since the new variables are not constrained by program execution. Thus, naturally, the suspension behavior of the original program is preserved.

Theorem 5.1. Let $\mathcal{G}, \mathcal{G}'$ be confluent transition systems for s and P, and let $\mathcal{D}$ be a nonfailing derivation of $\mathcal{G}$. Then,

(1) *if $\mathcal{D}$ is finite, then there is a finite nonfailing derivation of $\mathcal{G}'$ isomorphic to $\mathcal{D}$; and*

(2) *if $\mathcal{D}$ is infinite and $\mathcal{G}$ and $\mathcal{G}'$ are fair, then there is an infinite derivation of $\mathcal{G}'$ isomorphic to $\mathcal{D}$.*

PROOF. Assume that $\mathcal{D} = s_1 \xrightarrow{A_1;C_1} s_2 \xrightarrow{A_2;C_2} \cdots$ is a derivation of $\mathcal{G}$, where $s_1 = s$, and that sel' is the selection function of $\mathcal{G}'$ (see Definition 5). In the case where $\mathcal{D}$ is finite, we do not use fairness in this proof. We inductively construct a sequence of partial derivations $\mathcal{D}'_n$ in $\mathcal{G}'$ whose limit $\mathcal{D}' = s'_1 \rightarrow s'_2 \rightarrow \cdots$ is isomorphic to $\mathcal{D}$. Simultaneously, we construct a sequence of functions $f_n : \{0, \ldots, n\} \rightarrow I\!N$, each an extension of its predecessor, such that $\mathcal{D}'_n$ and f_n satisfy $\mathcal{D}'_n = s'_1 \xrightarrow{A_{f_n(1)};C_{f_n(1)}} s'_2 \xrightarrow{A_{f_n(2)};C_{f_n(2)}} \cdots \xrightarrow{A_{f_n(n)};C_{f_n(n)}} s'_{n+1}$. (Subscripts on f are often elided below.)

basis: In step zero of this construction, define $s'_1 = s_1$ and $f(0) = 0$. For uniformity in the step below, it is convenient to define $s'_0 = s_0 = \langle \emptyset, \emptyset \rangle$ and view transition zero as introducing all the atoms of s in each derivation. A_0 and C_0 are not needed and undefined. Observe that $con(s'_1)$ is consistent by assumption on $\mathcal{D}$.

step: In step $k \geq 1$ of the construction, we have the following induction assumptions:

1. $\mathcal{D}'_{k-1}$ is a partial deduction with $con(s'_k)$ consistent;
2. $\mathcal{D}'_{k-1} = s'_1 \xrightarrow{A_{f(1)};C_{f(1)}} s'_2 \xrightarrow{A_{f(2)};C_{f(2)}} \cdots \xrightarrow{A_{f(k-1)};C_{f(k-1)}} s'_k$.

We must extend $\mathcal{D}'_{k-1}$ to $\mathcal{D}'_k$ and f_{k-1} to f_k. For $1 \leq i < k, f_k(i) =_{df} f_{k-1}(i)$. Thus, eliding the subscript, we must determine $f(k)$.

From induction assumption (2), it follows that the least $l \in I\!N$ such that $l \notin \{f(i) \mid 0 \leq i < k\}$ has $con(s'_k) \leq con(s_l)$. This implies $\neg stuck^P_{\mathcal{CF}}(A_l, s'_k)$, from which it follows that $sel'(s'_k)$ is not empty, by definition of confluent transition system (Definition 1) and $sched^P$ (Definition 3). Fixing any $A \in sel'(s'_k)$, we have $\neg stuck^P_{\mathcal{CF}}(A, s'_k)$.

Again, from induction assumption (2), it follows that $con(s_{m+1}) \leq con(s'_k)$, where $m = max\{f(i) \mid 0 \leq i < k\}$. We have $\neg stuck^P_{\mathcal{CF}}(A, con(s'_k))$ from the assumption that $A \in sel'(s'_k)$, so it follows from the definition of $stuck^P_{\mathcal{CF}}$ (Definition 1) that we have $\neg stuck^P_{\mathcal{CF}}(A, con(s_{i+1}))$ for all $i \geq m$.

Because each atom contains at least one variable, renaming ensures that each atom in a derivation is introduced at most once. Since $A \in goal(s'_k)$, we can fix the $q < k$ such that A is introduced by the transition from s'_q to s'_{q+1}. By the induction assumption, it follows that A appears in $C_{f(q)}$. Hence, A is introduced by the transition in $\mathcal{D}$ from $s_{f(q)}$ to $s_{f(q)+1}$. Since $\neg stuck^P_{\mathcal{CF}}(A, con(s_{i+1}))$ for all $i \geq m$ and either $\mathcal{G}$ is fair or $\mathcal{D}$ is finite, there exists $n > f(q)$ such that $A = A_n$ (A is the atom reduced in the nth step of $\mathcal{D}$). Again, since each atom contains at least one variable, n is unique. Define $f(k) = n$.

We show that the induction hypotheses are preserved by the construction step. First, observe that $C_n \ll_{s'_k} P$ since the variables of C_n do not occur in s_1 or in any C_i where $i \neq n$. Now, recall that $s_n \xrightarrow{A_n;C_n} S_{n+1}$ and that $m = max\{f(i) \mid 0 \leq i < k\}$. By assumption on $\mathcal{D}$, $con(s_{max(n,m)+1})$ is consistent. From induction assumption (2) and the choice of $f(k) = n$, we have $con(s_{max(n,m)+1}) \leq con(s'_{k+1})$. It follows that $con(s'_{k+1})$ is consistent, and hence (by Definition 1) that $reduce_{\mathcal{CF}}(s'_k, s'_{k+1}, A_{f(k+1)})$. It now follows from the definitions of transition system and partial derivation (Definitions 5 and 6) that $s'_k \xrightarrow{A_n;C_n} s'_{k+1}$, i.e., that $\mathcal{D}'_k = s'_1 \xrightarrow{A_{f(1)};C_{f(1)}} s'_2 \xrightarrow{A_{f(2)};C_{f(2)}} \cdots \xrightarrow{A_{f(k)};C_{f(k)}} s'_{k+1}$ is a partial derivation. Thus, both induction hypotheses are preserved by the step.

Now, we show that f is a bijection. Because each atom contains a variable, no atom is introduced more than once or reduced more than once in either $\mathcal{D}$ or $\mathcal{D}'$. So, by the way $f(k)$ is chosen in the construction step, f is injective.

To see that f is surjective, assume for contradiction that there is a $j \in \mathbb{N}$ such that $\forall i. f(i) \neq j$. (Thus, A_j is never reduced in $\mathcal{D}'$.) Fix the least such j. Consider the $q < j$ such that A_j is introduced by the transition from s_q to s_{q+1}. (That is, A_j appears in C_q.) By the minimality of j, there exists q' such that $f(q') = q$. Thus, $A_j \in goal(s'_{q'+1})$. Again, by the minimality of j, there exists an m such that $\{1, \ldots, j-1\} \subseteq \{f(l) \mid 1 \leq l \leq m\}$. Fixing any such m, we have $con(s'_{m+1}) \leq con(s_j)$. Thus, since $stuck^P_{\mathcal{CF}}(A_j, con(s_j))$, we have $stuck^P_{\mathcal{CF}}(A_j, con(s'_{m+1}))$. It follows that the assumption that A_j is never reduced in $\mathcal{D}'$ contradicts either the fairness of $\mathcal{G}'$ or the finiteness of $\mathcal{D}$. □

We prove Theorems 5.2 and 5.3 simultaneously. The construction resembles that of the previous proof, and is presented somewhat less formally.

Theorem 5.2. If $s \in State_C$ and $P \in Prog$ lead to suspension, then any confluent transition system for s and program P suspends.

Theorem 5.3. If $s \in State_C$ and $P \in Prog$ lead to local suspension, then any fair confluent transition system for s and P locally suspends.

PROOF. Assume that $\mathcal{D} = s_1 \xrightarrow{A_1;C_1} s_2 \xrightarrow{A_2;C_2} \cdots$ is a locally suspending derivation of $\mathcal{G}$, where $s_1 = s$, and that sel' is the selection function of $\mathcal{G}'$ (see Definition 5). In the case where $\mathcal{D}$ has the stronger property of globally suspending, we do not use fairness in this proof.

We inductively construct $\mathcal{D}' = s'_1 \rightarrow s'_2 \rightarrow \cdots$ in $\mathcal{G}'$, and show that $\mathcal{D}'$ locally suspends. Intuitively, $\mathcal{D}'$ performs a subset of the transition steps taken by $\mathcal{D}$, although in an order dictated by sel'. Simultaneously, we construct a partial

injection $f : \mathbb{N} \to \mathbb{N}$ such that $\mathcal{D}' = s'_1 \overset{A_{f(1)};C_{f(1)}}{\longrightarrow} s'_2 \overset{A_{f(2)};C_{f(2)}}{\longrightarrow} \ldots$, which is used after the construction is complete to demonstrate that $\mathcal{D}'$ locally suspends. The function f may not be total in the case that $\mathcal{D}'$ globally suspends and hence is finite. $\mathcal{D}'$ can be shorter than $\mathcal{D}$ because the confluent scheduling relation is more strict than the standard scheduling relation.

basis: In step zero of this construction, define $s'_1 = s_1$ and $f(0) = 0$. For uniformity in the step below, it is convenient to define $s'_0 = s_0 = \langle \emptyset, \emptyset \rangle$ and view transition zero as introducing all the atoms of s in each derivation. A_0 and C_0 are not needed and undefined. Observe that $con(s'_1)$ is consistent by assumption on $\mathcal{D}$.

step: In step k of the construction, inductively assume that $\mathcal{D}'$ is partially constructed up to step $k-1$, that the partial derivation has the form

$$s'_0 \to s'_1 \overset{A_{f(1)};C_{f(1)}}{\longrightarrow} s'_2 \overset{A_{f(2)};C_{f(2)}}{\longrightarrow} \cdots \overset{A_{f(k-1)};C_{f(k-1)}}{\longrightarrow} s'_k,$$

and that $con(s'_k)$ is consistent.

If $sel'(s'_k)$ is empty, then the constructions of $\mathcal{D}'$ and f are complete (i.e., for all $i \geq k$, $f(i)$ and s_{i+1} are undefined), in which case $\mathcal{D}'$ suspends. Otherwise, fix $A \in sel'(s'_k)$ and define $f(k)$ and s'_{k+1} as follows.

We seek a suitable choice for the value of $f(k)$. From the induction assumption, it follows that $con(s_{m+1}) \leq con(s'_k)$, where $m = max\{f(i) \mid 0 \leq i < k\}$. We have $\neg stuck^P_{C\mathcal{F}}(A, con(s'_k))$ from the assumption that $A \in sel'(s'_k)$, so using the definition of $stuck^P_{C\mathcal{F}}$ (see Definition 1), we have $\neg stuck^P_C(A, con(s_{i+1}))$ for all $i \geq m$.

Because each atom contains at least one variable, renaming ensures that each atom in a derivation is introduced at most once. Since $A \in goal(s'_k)$, we can fix the $q < k$ such that A is introduced by the transition from s'_q to s'_{q+1}. By the induction assumption, it follows that A appears in $C_{f(q)}$. Hence, A is introduced by the transition in $\mathcal{D}$ from $s_{f(q)}$ to $s_{f(q)+1}$. Since $\neg stuck^P_C(A, con(s_{i+1}))$ for all $i \geq m$ and either $\mathcal{G}$ is fair or $\mathcal{D}$ is globally suspending, there exists $n > f(q)$ such that $A = A_n$ (A is the atom reduced in the nth step of $\mathcal{D}$. Again, since each atom contains at least one variable, n is unique. Define $f(k) = n$.

To show that the induction hypothesis is preserved by the construction step, we show the existence and consistency of s'_{k+1}, where $s'_k \overset{A_n;C_n}{\longrightarrow} s'_{k+1}$. First, observe that $C_n \ll_{s'_k} P$ since the variables of C_n do not occur in s_1 or in any C_i where $i \neq n$. This shows existence (see Definition 1). Now, recall that $s_n \overset{A_n;C_n}{\longrightarrow} s_{n+1}$ and that $m = max\{f(i) \mid 0 \leq i < k\}$. By assumption on $\mathcal{D}$, $con(s_{max(n,m)+1})$ is consistent. By construction, $con(s_{max(n,m)+1}) \leq con(s'_{k+1})$. It follows that $con(s'_{k+1})$ is consistent.

We now show that $\mathcal{D}'$ locally suspends. As observed in the construction, if $\mathcal{D}'$ is finite, it suspends, and hence locally suspends. Thus, assume $\mathcal{D}'$ is infinite. Because each atom contains a variable, no atom is introduced more than once or reduced more than once in either $\mathcal{D}$ or $\mathcal{D}'$. So, by the way $f(k)$ is chosen in the construction step, f is injective. It follows that $\mathcal{D}$ is infinite and, by definition of local suspension (Definition 3), that $\exists i. \exists A \in goal(s_i). \forall j. (j \geq i \Rightarrow stuck_C(A, con(s_i)))$. Fix such an A.

We show that it follows that $\mathcal{D}'$ is also locally suspending. Assume for contradiction that this is not the case. Then for each $i \in I\!N$, each atom introduced into $\mathcal{D}'$ at step i is eventually reduced since $\mathcal{G}'$ is fair by hypothesis. Using this fact, a simple induction on $I\!N$ shows that $\forall_j \in I\!N.\exists i \in I\!N.f(i) = j$, i.e., that f is surjective. Thus, if j' is the step at which A is introduced into $\mathcal{D}$, there is a step i' at which A is introduced into $\mathcal{D}'$. However, since there is no step j'' at which A is reduced in $\mathcal{D}\,(\forall j''.A \neq A_{j''})$, there is no step i'' at which A is reduced in $\mathcal{D}'$ $(\forall i''.A \neq A_{f(i'')})$. This yields the desired contradiction with the conclusion above that each atom added in $\mathcal{D}'$ is eventually reduced. □

Proposition 5.1. Let $s, s' \in State_C$. If there exist (fair) confluent transition systems for P with s and with s' which do not (locally) suspend; then there exists a (fair) confluent transition system for the composed state $s \parallel s'$ which does not (locally) suspend with P.

PROOF. We sketch the contrapositive for the case of suspension. The method can easily be adapted to the case of local suspension in a manner that is analogous to the other proofs in this Appendix.

Assume that there exists a confluent transition system $\mathcal{G}$ for $s_1 \parallel s_2$ and P which suspends. Let $\mathcal{D} = \bar{s}_1 \xrightarrow{A_1;C_1} \bar{s}_2 \xrightarrow{A_2;C_2} \cdots \bar{s}_n$ be a suspending derivation of $\mathcal{G}$.

We inductively define the notion of an atom A in the derivation $\mathcal{D}$ which is *derived from* the state s_k $k \in \{1, 2\}$ as follows. Let $\bar{s}_i$ be the first state of $\mathcal{D}$ in which A occurs. If $i = 1$, then A is derived from s_1 if $A \in goal(s_1)$ and from s_2 otherwise. If $i > 1$, then A is derived from s_k if A_{i-1} is derived from s_k.

Observe that $s_1 \parallel s_2 \sim s_2 \parallel s_1$. It is straightforward to inductively construct a transition system for P and $s_2 \parallel s_1$ that is a renaming of $\mathcal{G}$. Thus, we may assume without loss of generality that $\bar{s}_n$ contains an atom which is derived from s_1.

We inductively construct a suspending derivation $\hat{\mathcal{D}} = \hat{s}_1 \rightarrow \cdots \rightarrow \hat{s}_m$ for s_1 yielding the required contradiction.

Denote $\hat{s}_1 = s_1$. There are two cases: (1) $m = 1$; or (2) if some atom $A \in goal(\hat{s}_1)$ can be selected for confluent reduction, then it must be the case that A is a selected atom in $\mathcal{D}$ because, otherwise, A is in $goal(\bar{s}_n)$, and hence stuck while $con(\bar{s}_n) \leq con(\hat{s}_1)$, which would be a contradiction. Let $\bar{s}$ be the state of $\mathcal{D}$ in which A was selected for reduction. Since $con(\bar{s}) \leq con(\hat{s}_1)$ and $\bar{s} \neq fail$, A can be reduced in $\hat{s}_1$ with the same clause C applied to A in $\mathcal{D}$. Let $\hat{s}_2$ be the state resulting from the confluent reduction of $\hat{s}_1$ with selected atom and clause A and C.

In the ith step of the construction (given state $\hat{s}_i$), the argument is similar. If some atom $A \in goal(\hat{s}_i)$ can be selected, then it must be selected in some step of $\mathcal{D}$—because otherwise $A \in goal(\bar{s}_n)$ but $con(\bar{s}_n) \leq con(\hat{s}_i)$ (note that all previous reductions of $\hat{\mathcal{D}}$ involve the same selected atoms and clauses as steps in $\mathcal{D}$). Moreover, A can be reduced with the same clause $C = H : -Ask : Tell \mid B$ as in $\mathcal{D}$ because $con(\bar{s}_n) \neq fail$ and $con(\bar{s}_n) \leq con(\hat{s}_i) \wedge Ask \wedge Tell \wedge H = A$. □

Proposition 5.2. The confluent and standard transition semantics are equivalent for confluent programs.

PROOF. It is sufficient to prove that for confluent programs $\neg stuck_{\mathcal{C}}(A, \theta) \Rightarrow \neg stuck_{\mathcal{CF}}(A, \theta)$. Assume that $\neg stuck_{\mathcal{C}}(A, \theta)$. Thus, (a) $\exists C.try(A, \theta, C) = success$, or

(b) $\forall C.try(A,\theta,C) = fail$. If it is case (b), clearly, $\forall\theta' \leq \theta, try(A,\theta,C) = fail$, and so $\forall\theta' \leq \theta.\neg stuck_{\mathcal{C}}(A,\theta')$, from which $\neg stuck_{\mathcal{CF}}(A,\theta)$. If it is case (a), from Definition 3, we have that $\forall C'.try(A,\theta,C') \neq delay$. Thus, $\forall C'.try(A,\theta,C') \in \{success, fail\}$. Thus, $\forall\theta' \leq \theta, \forall C'.try(A,\theta,C') \in \{success, fail\}$. Thus, $\forall\theta' \leq \theta.\neg stuck_{\mathcal{C}}(A,\theta')$, and so $\neg stuck_{\mathcal{CF}}(A,\theta)$. □

Theorem 6.1. Let $\mathcal{G}$ be an abstract transition system for an abstract state s and a program P, and let s' be a concrete state such that $s \propto s'$. If $\mathcal{G}$ is fair and does not lead to (local) suspension, then any standard transition system for s' and P does not lead to (local) suspension.

PROOF. Here, we consider the more difficult case of local suspension; for the case of suspension, the proof is similar. Let us assume that $\mathcal{G}$ is a fair abstract transition system for s and P that does not locally suspend. We inductively construct a fair confluent transition system $\mathcal{G}'$ for s' and P that also does not locally suspend. Then the claim will follow by using Corollary 5.2. By using induction on the depth k of the nodes in $\mathcal{G}'$, we construct: (1) the concrete sel' function for the confluent transition system $\mathcal{G}'$, (2) a mapping f from the nodes of $\mathcal{G}'$ onto the nodes of $\mathcal{G}$ which shows the correspondence between the derivations of $\mathcal{G}'$ and the derivations of $\mathcal{G}$, and (3) a family of renamings $\rho_{s'_i}$ satisfying $f(s'_i) \propto_{\rho_{s'_i}} s'_i$ for each concrete state s'_i. Let s' be the source of $\mathcal{G}'$, where s' is any state such that $s \propto_\rho s'$. Then for every atom A such that $\neg stuck_{\mathcal{A}}(A,s')$, we have $\neg stuck_{\mathcal{CF}}(A,s)$ by definition of abstract stuckness and the fact that $con(s) \propto \rho(\tilde{\exists}_{goal(s')} con(s'))$. Thus, let $\rho_{s'} = \rho$, let $sel'(s')$ be given by $\rho(sel'(s')) = sel(s)$, and let $f(s') = s$.

Now, consider any derivation $s'_1 \overset{A_1;C_1}{\longrightarrow} s'_2 \overset{A_2;C_2}{\longrightarrow} \cdots s'_k$ in $\mathcal{G}'$ of length k. By the induction hypothesis, there exists a derivation $s_1 \overset{\rho_1(A_1;C_1)}{\longrightarrow} s_2 \overset{\rho_2(A_2;C_2)}{\longrightarrow} \cdots s_k$ in $\mathcal{G}$ such that $s'_i \propto_{\rho_i} s_i$ and $f(s'_i) = s_i$, $i = 1,\ldots,k$, and where ρ_i agrees with $\rho_{s'_i}$ on $Vars(s'_i)$ and also renames $Vars(C_i)$ appropriately. Also by the induction hypothesis, $sel(s_k) = \rho_k(sel'(s'_k))$. Let $A \in sel'(s'_k)$. Assume that $reduce_{\mathcal{CF}}(s'_k, s'_{k+1}, A)$. Then, since $s_k \propto_{\rho k} s'_k$, by the correctness requirement on $reduce_{\mathcal{A}}$, there exists $s_{k+1} \in State_{\mathcal{A}}$ such that $reduce_{\mathcal{A}}(s_k, s_{k+1}, \rho_k(A))$ and $s_{k+1} \propto_{\rho_{k+1}} s'_{k+1}$. We let $f(s'_{k+1}) = s_{k+1}, \rho_{s'_{k+1}} = \rho_{k+1}$, and $\rho_{k+1}(sel'(s'_{k+1})) = sel(s_{k+1})$.

Next, we show that $\mathcal{G}'$ does not locally suspend. Assume for contradiction that it does. Then there exists a derivation $s'_1 \overset{A_1;C_1}{\longrightarrow} s'_2 \overset{A_2;C_2}{\longrightarrow} \cdots s'_k \overset{A_k;C_k}{\longrightarrow} \cdots$ in $\mathcal{G}'$ such that $\exists i.\exists A \in goal(s'_i).\forall j.(j \geq i \Rightarrow stuck_{\mathcal{CF}}(A, con(s'_j)))$. Fix such an i and such an A. By construction,

$$f(s'_1) \overset{\rho_1(A_1;C_1)}{\longrightarrow} f(s'_2) \overset{\rho_2(A_2;C_2)}{\longrightarrow} \cdots f(s'_k) \overset{\rho_2(A_k;C_k)}{\longrightarrow} \cdots$$

is a derivation in $\mathcal{G}$ such that $\rho_i(A) \in goal(f(s'_i))$ and $\forall j.(j \geq i \Rightarrow stuck_{\mathcal{A}}(\rho_i(A), con(f(s'_j))))$, which contradicts the hypothesis that $\mathcal{G}$ does not locally suspend.

It remains to show that $\mathcal{G}'$ is fair. A similar contradiction can be derived by proving that if some derivation of $\mathcal{G}'$ has an atom that is not stuck, but that is never reduced, then there must be a corresponding atom in a corresponding derivation of $\mathcal{G}$ such that either the atom remains forever stuck, contradicting the assumption that $\mathcal{G}$ does not locally suspend, or else the atom becomes unstuck, but is never reduced, contradicting the assumption that $\mathcal{G}$ is fair. □

Michael Codish is grateful to the Departments of Computer Science at the University of Pisa and at KU Leuven where he was visiting while working on this paper. Moreno Falaschi was partially supported by the ESPRIT Basic Research Action 6707 ("Parforce"). William Winsborough was partially supported by NSF CCR-9210975.

REFERENCES

1. Brookes, S. D. and Roscoe, A. W., Deadlock Analysis in Networks of Communicating Processes, *Distributed Computing* 4:209–230 (1991).
2. Chandy, K. M. and Misra, J., Deadlock Absence Proofs for Networks of Communicating Processes, *Information Processing Lett.* 9(4):185 (1979).
3. Codish, M., Falaschi, M., and Marriott, K., Suspension Analysis for Concurrent Logic Programs, in: K. Furukawa (ed.), *Proc. 8th Int. Conf. on Logic Programming*, MIT Press, Cambridge, MA, 1991, pp. 331–345.
4. Codish, M., Falaschi, M., and Marriott, K., Suspension Analyses for Concurrent Logic Programs, *ACM Trans. Programming Languages and Syst.* 16(3):649–686 (1994).
5. Codish, M., Falaschi, M., Marriott, K., and Winsborough, W., Efficient Analysis of Concurrent Constraint Logic Programs, in: A. Lingas, R. Karlsson, and S. Carlsson (eds.), *Proc. 20th Int. Colloquium on Automata, Languages, and Programming*, vol. 700 of *LNCS*, Springer-Verlag, Berlin, 1993, pp. 633–644.
6. Codognet, C., Codognet, P., and Corsini, M., Abstract Interpretation for Concurrent Logic Languages, in: S. Debray and M. Hermenegildo (eds.), *Proc. North American Conf. on Logic Programming'90*, MIT Press, Cambridge, MA, 1990, pp. 215–232.
7. Codognet, C. and Codognet, P., Guarded Constructive Disjunction: Angel or Demon?, in: [20], pp. 345–361.
8. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Proc. 4th ACM Symp. Principles of Programming Languages*, ACM, 1977, pp. 238–252.
9. Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C., Compositional Analysis for Concurrent Constraint Programming, in: *Proc. 8th IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 210–221.
10. Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C., Confluence and Concurrent Constraint Programming, in: V. S. Alagar and M. Nivat (eds.), *Proc. 4th Int. Conf. on Algebraic Methodology and Software Technology (AMAST'95)*, vol. 936 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995, pp. 531–545.
11. Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, NJ, 1985.
12. Horiuchi, K., Less Abstract Semantics for Abstract Interpretation of FGHC Programs, in: *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan, 1992, pp. 897–906.
13. Janssens, G. and Bruynooghe, M., Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation, *J. Logic Programming* 13:205–258 (1992).
14. Lassez, J.-L. and Maher, M. J., Closures and Fairness in the Semantics of Programming Logic, *Theoretical Comput. Sci.* 29:167–184 (1984).
15. Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, Berlin, 1987.
16. Marriott, K., Garcia de la Banda, M., and Hermenegildo, M., Analyzing Logic Programs with Dynamic Scheduling, in: *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.
17. Marriott, K. and Oderski, M., A Confluent Calculus for Concurrent Constraint Programming with Guarded Choice, in: [20], pp. 310–327.
18. Milner, R., *Communication and Concurrency*, Prentice-Hall Int., UK, 1989.

19. Mishra, P., Towards a Theory of Types in Prolog, in: *Int. Symp. on Logic Programming*, IEEE Computer Society Press, 1984, pp. 289–298.
20. Montanari, U. and Rossi, F. (eds.), *Proc. 1st Int. Conf. on Principles and Practice of Constraint Programming (CP'95)*, vol. 976 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995.
21. Naish, L., Heterogeneous SLD Resolution, *J. Logic Programming* 1(4) (1984).
22. Peng, W. and Purushothaman, S., Data Flow Analysis of Communicating Finite State Machines, *ACM Trans. Programming Languages and Syst.* 13(2):399–442 (1991).
23. Saraswat, V., Rinard, M., and Panangaden, P., Semantic Foundation of Concurrent Constraint Programming, in: *Proc. 18th Annu. ACM Symp. on Principles of Programming Languages*, ACM, 1991.
24. Saraswat, V. A., Concurrent Constraint Programming Languages, Ph.D. Thesis, Carnegie Mellon University, Jan. 1989. Also in *ACM Distinguished Dissertation Series*, MIT Press, 1993.
25. Sato, T. and Tamaki, H., Enumeration of Success Patterns in Logic Programs, *Theoretical Computer Science* 34:227–240 (1984).
26. Shapiro, E. Y., The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys* 21(3):412–510 (1989).
27. Van Hentenryck, P., Cortesi, A., and Le Charlier, B., Type Analysis of Prolog Using Type Graphs, in: *Proc. ACM SIGPLAN'94 Conf. on Programming Language Design and Implementation*, ACM, June 1994, pp. 337–348.
28. Yang, R. and Aiso, H., P-Prolog: A Parallel Language Based on Exclusive Relation, in: E. Y. Shapiro (ed.), *Proc. 3rd Int. Conf. on Logic Programming*, vol. 225 of *LNCS*, Springer-Verlag, Berlin, 1986, pp. 255–269.
29. Yelick, K. and Zachary, J., Moded Type Systems for Logic Programming, in: *Proc. 16th Annu. ACM Symp. on Principles of Programming Languages*, ACM, 1989, pp. 116–124.
30. Zaffanella, E., Levi, G., and Giacobazzi, R., Abstracting Synchronization in Concurrent Constraint Programming, in: *Proc. 5th Int. Symp. on Programming Language Implementation and Logic Programming*, vol. 844 of *LNCS*, Springer-Verlag, Berlin, 1994, pp. 57–72.
31. Zaffanella, E., Domain Independent Ask Approximation in ccp, in: [20], pp. 362–379.